CMPSC 465 Spring 2024 | Data Structures & Algorithms | Mehrdad Mahdavi and David Koslicki

Worksheet 12

Wednesday, April 10, 2024

1. **Bellman-Ford.** Let $d(i,k)$ be the length of the shortest path from s to i that uses at most k edges. Then the updating procedure of Bellman-Ford can be expressed with the following recurrence relation: $d(i,k) = min(d(i,k-1), d(j,k-1) + \ell(j,i))$ for all $j \in V$ such that $(j,i) \in E$. This means that every node reachable from s in at most k steps must be one step away from some node reachable in at most $k-1$ steps. Furthermore, the shortest path to that node in at most k steps is the smallest among the paths from its neighbors reachable in $k-1$ steps. The table representing this dynamic program has indices i and k, both of which are bounded by $|V|$ (remember that a path cannot be longer than $|V|-1$ nodes). However, every incoming edge must be considered for each cell in the table (therefore every edge in the graph is considered once per time k is increased), so assuming $|E| > |V|$, the running time of this implementation is still $O(|V||E|)$.

2. **Unique Paths.** Since the robot can only move down or right, there's only one way to reach the cells in the top column (all moves are down). Similarly, there is only one way to reach the cells in the top row (all moves are right).

   What about the inner cells?

   You can reach the cell at (i, j) by making a down move from the upper cell $(i-1, j)$ or making a right move from the left cell $(i, j-1)$. Thus, the number of unique ways to reach (i, j) is sum of unique ways to reach $(i-1, j)$ and $(i, j-1)$.

   Recursive approach:

   (a) Let uniquePaths be your function name that starts with m and n as input parameters

   (b) Check the base case: if m or n is 1 than return 1(top column or top row)

   (c) Otherwise return the sum of uniquePaths(m-1,n) and uniquePaths(m,n-1)

   The recursive approach is inefficient as it calculates same sub-problems multiple times. This approach is presented only for learning purpose and we don't expect you to write it in your exams. Just the dynamic programming approach will be sufficient.

   Dynamic Programming approach:

   (a) Initialize a 2D array d[m][n] = number of paths and set all values in this array to 1.

   (b) Iterate over all inner cells and update their values:
   d[col][row] = d[col-1][row] + d[col][row-1]

   (c) Your final answer is d[m-1][n-1]

3. **Pebbling a Checkerboard.**

(a) There are 8 possible patterns: the empty pattern, the 4 patterns which each have exactly one pebble, and the 3 patterns that have exactly two pebbles (on the first and fourth squares, the first and third squares, and the second and fourth squares).

(b) Number the 8 patterns 1 through 8, and define $S \subseteq \{1,2,\ldots,8\} \times \{1,2,\ldots,8\}$ to be all (a,b) such that pattern a is compatible with pattern b. For each pattern, there are a constant number of patterns that are compatible (for example, every pattern is compatible with the empty pattern).

Sub-problems and Recursion: We consider the sub-problem $L[i,j]$, $i = 0,1,2,\ldots,n$ and $j \in \{1,2,\ldots,8\}$ to be the maximal value achievable by pebbling columns $1,2,\ldots,i$ such that the final column has pattern j. Also let $V(i,j)$ be the value gained by placing pattern j on column i. Then L has the following recursion:

$$L[i+1,j] = \max_{(k,j)\in S} L[i,k] + V(i+1,j)$$

The base case is $L[0,j] = 0$ for all j. In order to recover the optimal placement, we should also maintain a back-pointer: $P[i+1,j]$ is the value of k such that $(k,j) \in S$ and $L[i,k]$ is maximal.

Algorithm and Running Time: For $i = 0,1,\ldots,n$, for $j = 1,2,\ldots,8$, compute $L[i,j]$ and $P[i,j]$ using the recurrence. Note that $L[i,j]$ and $P[i,j]$ can be computed using the recursion in constant time since we only need to check a constant number of possible k. The value of the optimal placement is given by $\max_j L[n,j]$.

To recover the optimal placement, let j^* be the value of j for which $L[n,j]$ is maximal. Then, column n should be pebbled using pattern j^*. Then, column $n-1$ should be pebbled using pattern $P[n,j^*]$, column $n-2$ with pattern $P[n-1,P[n,j^*]]$, and so on.

The running time of this algorithm is $O(n)$ because computing the recurrence takes $O(n)$ time and backtracking takes $O(1)$ time per column.

4. Change Making.

(a) For $0 \le w \le W$, define

$$f(w) = \text{the minimum number of coins needed to make a change for } w.$$

It satisfies

$$f(w) = \min \begin{cases} 0 & \text{if } w = 0, \\ 1 + \min_{j:\, v_j \le w} f(w - v_j) & \\ \infty & \end{cases}$$

The answer is $f(W)$, where ∞ means impossible. So, the final dynamic programming algorithm is to initialize an array f of length $W+1$ with $f[0] = 0$, then iterate from $i = 1$ to $i = W$, calculating $f[i]$ for each index. It takes $O(nW)$ time since calculating $f[i]$ takes $O(n)$.

(b) For $0 \le i \le n$ and $0 \le w \le W$, define

$$f(i,w) = \text{the minimum number of coins (among the first } i \text{ coins) needed to make a change for } w, \text{ having one coin per denomination.}$$

It satisfies

$$f(i,w) = \min \begin{cases} 0 & \text{if } i = 0 \text{ and } w = 0, \\ f(i-1,w) & \text{if } i > 0 \text{ and } v_i > w, \\ 1 + f(i-1, w-v_i) & \text{if } i > 0 \text{ and } v_i \leq w, \\ \infty \end{cases}$$

The second case is saying, we didn't select the i^{th} coin; thus, we should make change for w from the first $i-1$ coins.

The third case is saying, we selected the i^{th} coin; therefore, we should change $w - v_i$ with the first $i-1$ coins.

The answer to the problem is $f(n,W)$. So, the final dynamic programming algorithm is to initialize a matrix f of length $n+1$ by $W+1$, then iterate from $i = 0$ to $i = n$ for $w = 0$, calculating $f[i,0]$ for each index. Then repeat for $w = 1$ and so on. It takes $O(nW)$ time since calculating each entry takes $O(1)$ and there are $O(nW)$ entries.

5. **Rod Cutting.** A naive solution to this problem is to generate all configurations of different pieces and find the highest-priced configuration. This solution is exponential in terms of time complexity.

   (i) Optimal Substructure: We can get the best price by making a cut at different positions and comparing the values obtained after a cut. We can recursively call the same function for a piece obtained after a cut. Let cutRod(n) be the required (best possible price) value for a rod of length n. cutRod(n) can be written as follows.
   cutRod(n) = max(price[i] + cutRod(n-i)) for all i in $\{1, \ldots, n-1\}$

   (ii) Overlapping substructure: Many subproblems are solved repeatedly. Since the same subproblems are called again, this problem has the Overlapping Subproblems property. So the Rod Cutting problem has both properties of a dynamic programming problem. Like other typical Dynamic Programming problems, recomputations of the same subproblems can be avoided by constructing a temporary array to store results of subproblems.

   So, the final dynamic programming algorithm is to initialize an array A of length $n+1$ with $A[0] = 0$, then iterate from $i = 1$ to $i = n$, calculating cutRod(i) for each index and storing the result in $A[i]$. The final answer is $A[n]$, which can be computed in $O(n^2)$.